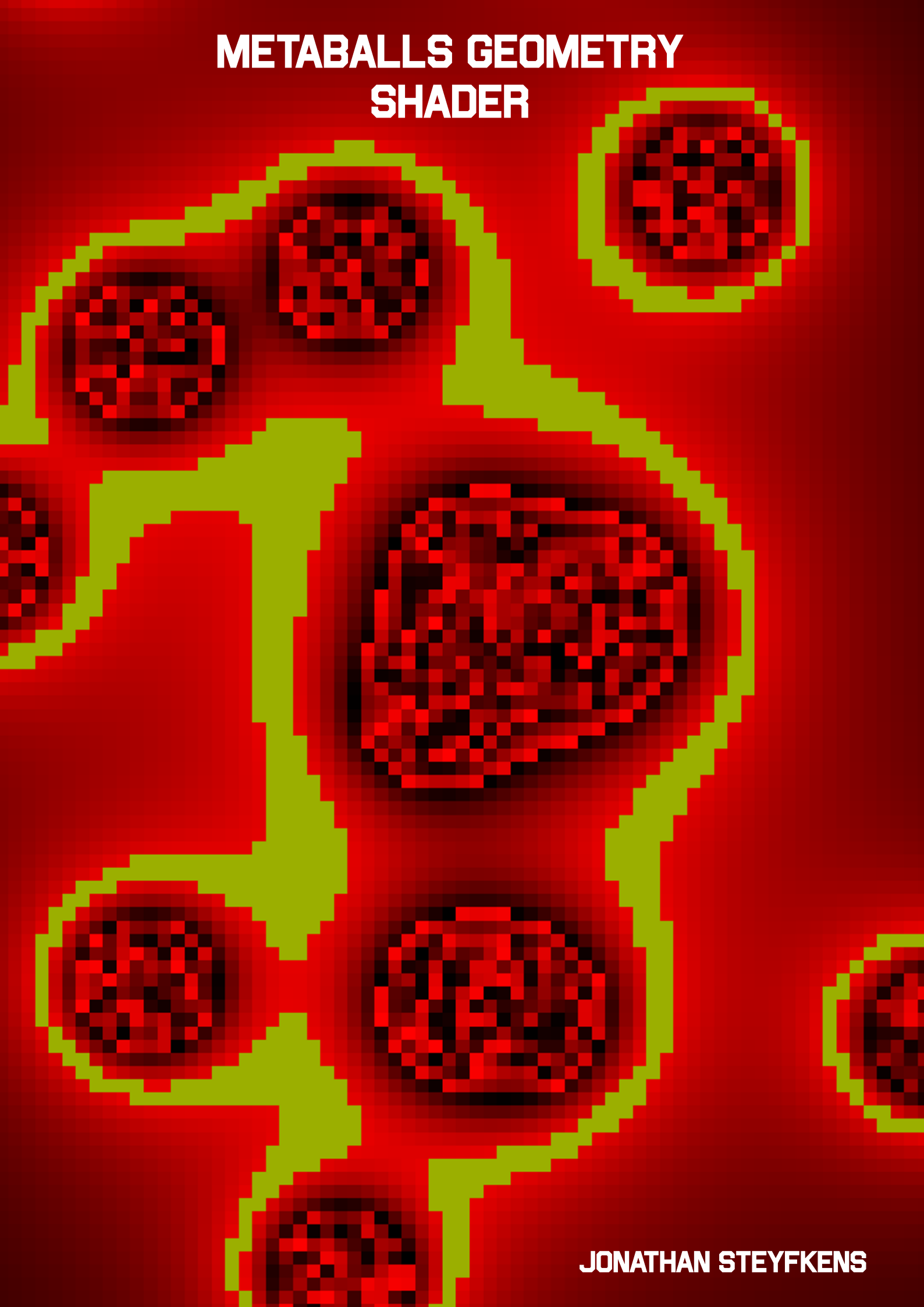
METABALLS GEOMETRY SHADER



JONATHAN STEYFKENS

Introduction

First I'll briefly explain what metaballs are and show an implementation in both 2D and 3D. Metaballs are geometric shape that represents a force field. When these balls come close to each other they will start attracting each other which causes them to melt together into 1 ball. Metaballs are often used for simulating fluids although they can also be used for other purposes. Some people tend to use them as a way to quickly block out a shape in blender for sculpting. Metaballs or blobs, are used in Portal 2 as a special puzzle element.

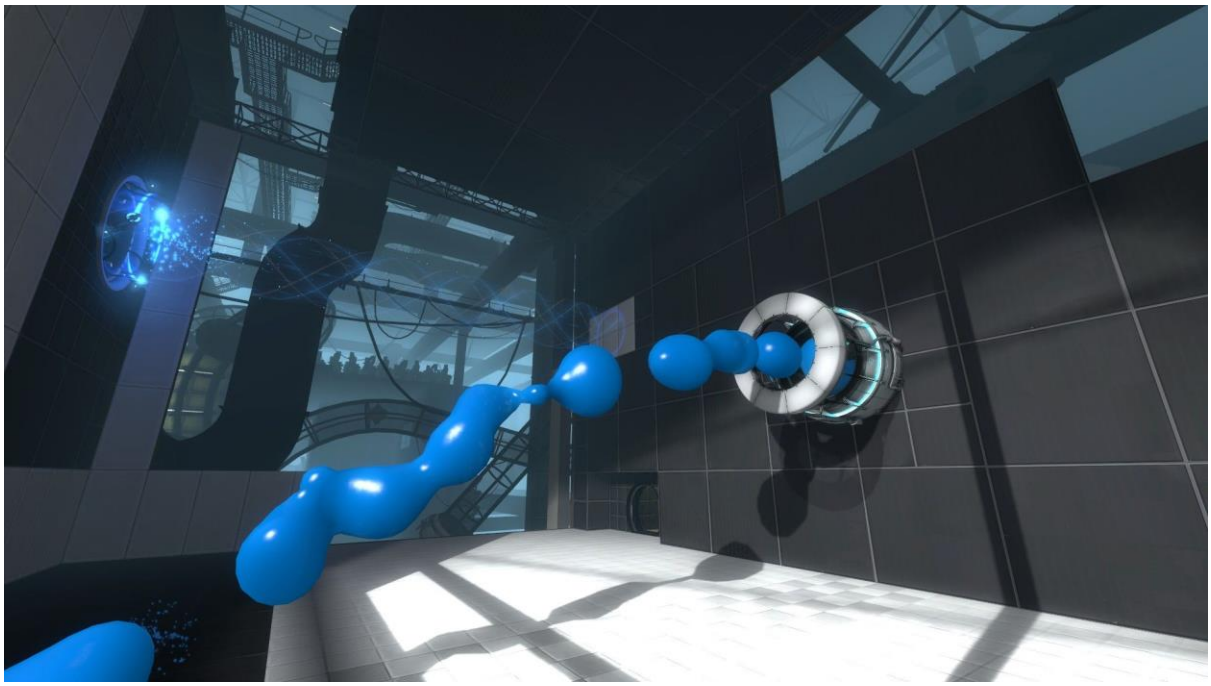


Figure 1

Back to the 2D age

In 2D we are actually drawing circles. The formula of a circle is quite easy. X_i and Y_i represent the centre of our circle.

$$(x - x_i)^2 + (y - y_i)^2 = r^2$$

Figure 2

To determine if the point (x, y) is on the circle we have to reform our equation.

$$\frac{r^2}{(x - x_i)^2 + (y - y_i)^2} = 1$$

Figure 3

The biggest problem with this equation is that it has 2 parameters, x and y. These correspond with a pixel position on the screen. You might be wondering why this is important. To calculate whether a cell is inside a metaball we have to take a summation of this formula for each metaball centre and radius. If our result of this summation for the current pixel/grid cell is smaller or equal the 1 this means, we are inside a metaball (colour the pixel) else we are outside of a ball.

$$\sum_{i=1}^{n} \frac{r^2}{(x - x_i)^2 + (y - y_i)^2} = 1$$

Figure 4

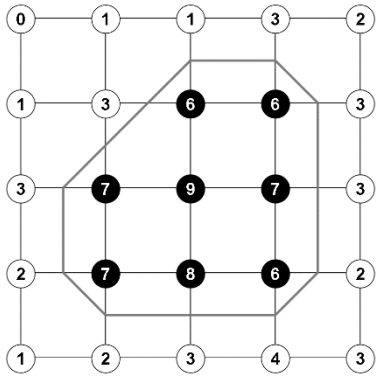


Figure 5

Doing this for every pixel is slow. Using an algorithm called marching squares is a good solution to make it faster. Instead of checking every pixel we divide our screen into grid cells and calculate what cell vertex is in a metaball and which one is not (F.4). Then we look up our grid case in a table and decide how to layout our drawing. This is the basic ideas about metaballs. The next chapter will extend this idea to 3D.

Extending to 3D

In 3D it's a bit harder to represent every point in the space. Every pixel doesn't correspond with a "cell" anymore. This makes it a bit harder and we will have to define our grid ourselves. A vertex buffer with evenly spaced vertices will make up grid. The formula to calculate the force in a given point is very similar to the formula used for 2D:

$$\sum_{i=1}^{n} \frac{r_p^2}{||(x - x_p)||^2} = 1$$

Figure 6

x_p represents the centre of our current metaball and r_p represents the radius of our metaball. Each metaball is put into a vector containing the position of the metaball and the force radius. The hardest part here is visualizing the metaballs in our 3D space. This can be achieved with Marching Cubes.

Marching Cubes

Marching cubes is often used to create a simplified 3D mesh. It works by checking at what points the edges of our cube intersect the surface we define.

Different cases of marching cube. The black vertices are intersecting with our shape.

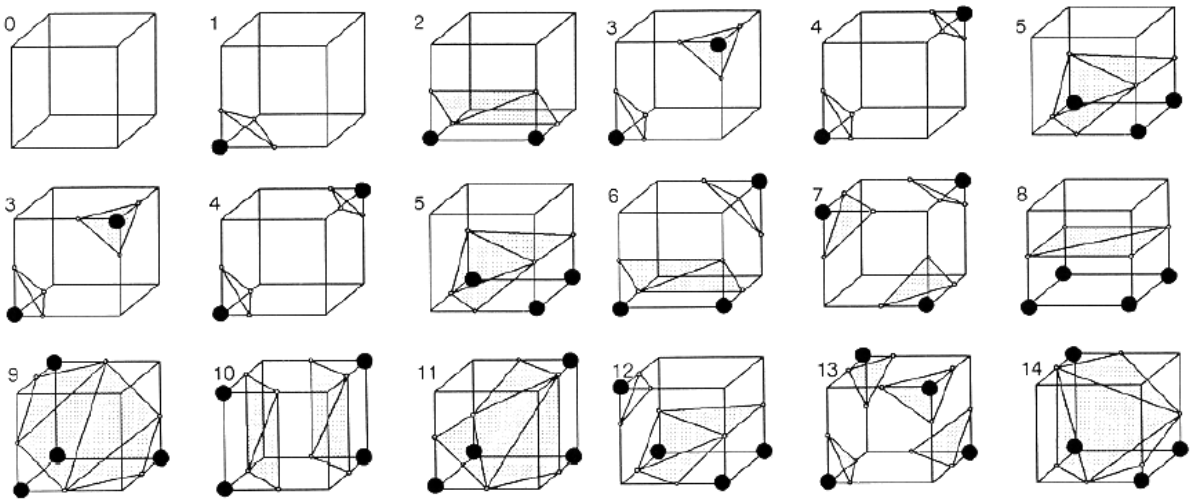


Figure 7

Unfortunately, a geometry shader cannot receive 8 vertices as input (maximum is 6). Instead split up a cube into 6 congruent tetrahedrons and pass 4 vertices each to the geometry shader. This will result in a different kind of lookup table and a slightly different algorithm.

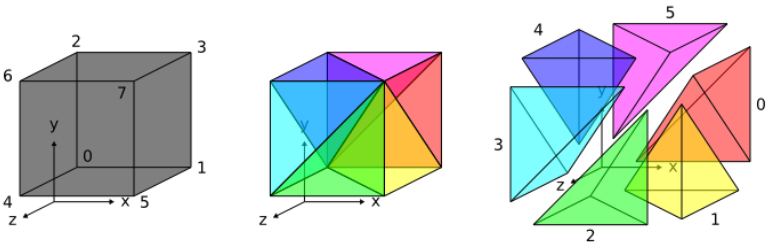


Figure 8

Vertex Shader

```
SampleData SampleFieldVS(float4 Pos : POSITION){
	SampleData output = (SampleData)0;

	float3 worldPos = mul(float4(Pos.xyz,1),gMatrixWorld).xyz;

	output.Field = 0;
	for(int i = 0; i < gNumMetaballs; i++)
	{
		output.Field += CalculateMetaball(worldPos,gMetaballs[i],gMetaballs[i].w);
	}
	// Do metaball calculations and add it to field
	// Currently transform in geometry shader;
	output.Pos = mul(float4(Pos.xyz,1),gMatrixWorldViewProj);
	output.Field.xyz = -normalize(mul(output.Field.xyz,gMatrixViewIT));
	return output;
}
```

Figure 9

Our vertex shader contains the code to calculate our necessary data. It outputs a SampleData struct which contains a “Position” and a “Field” (float4). The fields xyz components contain the direction of the force, the w component contains the size. For every metaball in our scene we call a function “CalculateMetaball”. This function calculates the formula seen at the start of this chapter. The metaball is made up of 1 float4(xyz is the position, w is size of the force field).

```
float4 CalculateMetaball(float3 pos, float3 center, float radiusSquared)
{
	float4 o;
	float dist = length(pos - center);
	// Calculate our w( force) by using our formula seen in the paper
	float result = radiusSquared / (dist*dist);

	// Our field direction is basically our direction
	o.xyz = -(pos-center);
	o.w = result;
	return o;
}
```

Figure 10

I strived for simplicity in this function instead of performance. Calculating the root in the length function isn’t very performant. See the references for faster methods.

Geometry Shader

With marching tetrahedrons, we have 16 different cases, 4 vertices that have 2 different states (inside or outside).

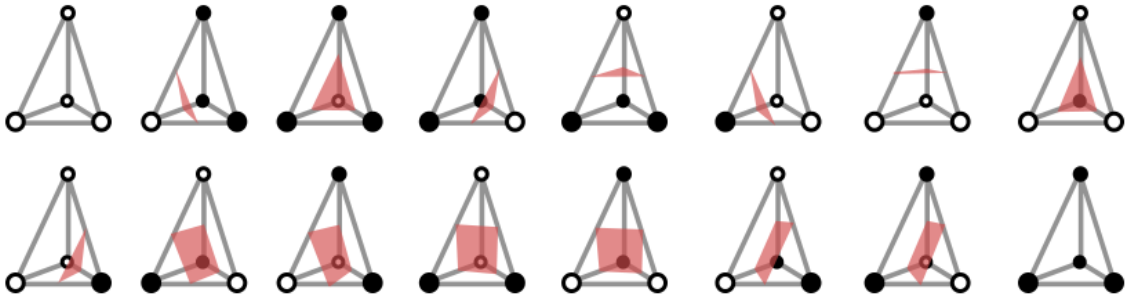
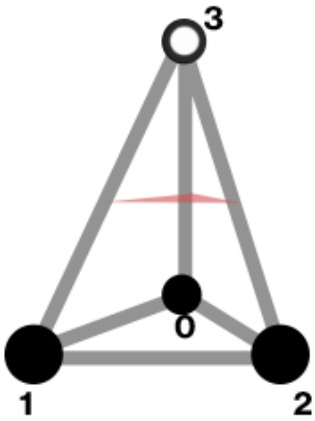


Figure 11

All these cases are stored inside an edge table.

```
const TetrahedronIndices EdgeTableGS[16] =
  {
    { 0, 0, 0, 0 }, { 0, 0, 0, 0 },
    { 3, 0, 3, 1 }, { 3, 2, 0, 0 },
    { 2, 1, 2, 0 }, { 2, 3, 0, 0 },
    { 2, 0, 3, 0 }, { 2, 1, 3, 1 },
    { 1, 2, 1, 3 }, { 1, 0, 0, 0 },
    { 1, 0, 1, 2 }, { 3, 0, 3, 2 },
    { 1, 0, 2, 0 }, { 1, 3, 2, 3 },
    { 3, 0, 1, 0 }, { 2, 0, 0, 0 },
    { 0, 2, 0, 1 }, { 0, 3, 0, 0 },
    { 0, 1, 3, 1 }, { 0, 2, 3, 2 },
    { 0, 1, 0, 3 }, { 2, 1, 2, 3 },
    { 3, 1, 2, 1 }, { 0, 1, 0, 0 },
    { 0, 2, 1, 2 }, { 0, 3, 1, 3 },
    { 1, 2, 3, 2 }, { 0, 2, 0, 0 },
    { 0, 3, 2, 3 }, { 1, 3, 0, 0 },
    { 0, 0, 0, 0 }, { 0, 0, 0, 0 },
  };
```

Every line represents 4 vertices. Each pair shows us between what 2 vertices we have to cut.




```
{ 0, 3, 2, 3 }, { 1, 3, 0, 0 },
```

This line will produce the following plane.
After we have found our plane we calculate our intersection point by interpolating the field variable

```
SurfaceVertex CalculateIntersection(SampleData a, SampleData b)
{
	SurfaceVertex output = (SurfaceVertex)0;
	// calculate interpolation
	float t = (2.0 - (a.Field.w + b.Field.w)) / (b.Field.w - a.Field.w);

	output.Pos = 0.5 * (t*(b.Pos - a.Pos) + (b.Pos + a.Pos));

	//output.Pos = (a.Pos + b.Pos)/2.0f;
	output.N = 0.5 * (t*(b.Field.xyz - a.Field.xyz) + (b.Field.xyz + a.Field.xyz));

	return output;

}
```

First calculate an interpolation constant using our Field strength variable. Then we apply this using the default linear interpolation.

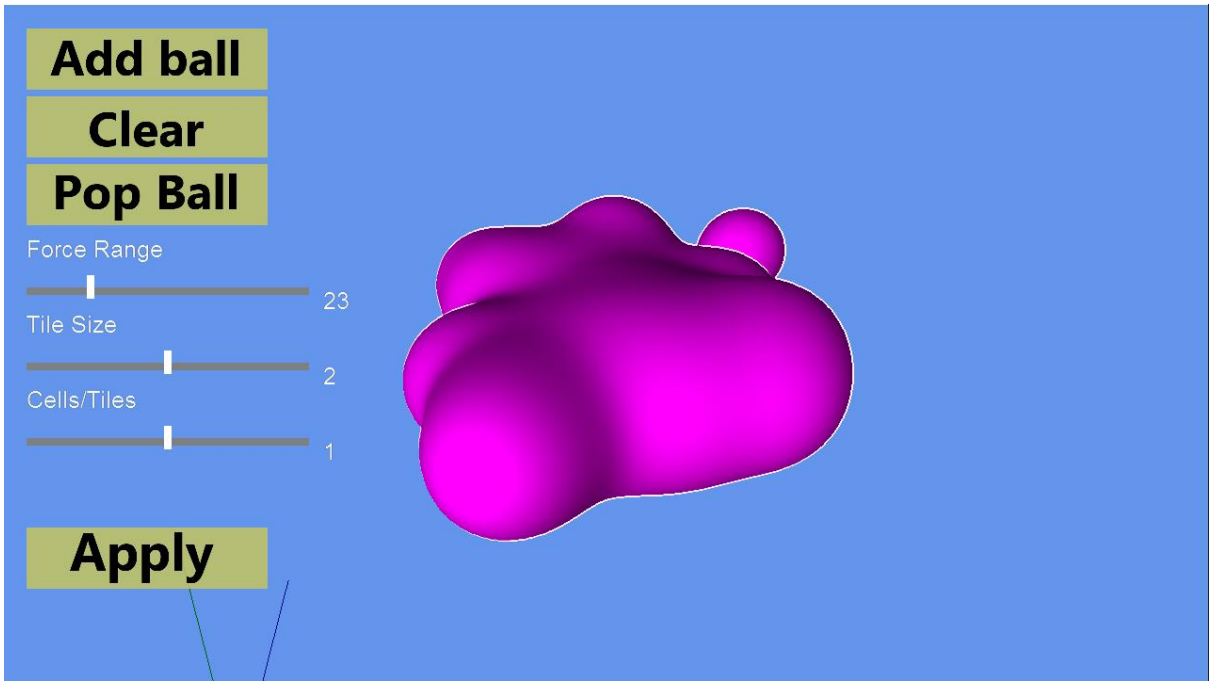
To get the right case from our edge table a binary number is created (0 for inside and 1 for outside) using our 4 vertices. Ex. 0001 -> last vertex is outside the mesh. The max value is 15 and min is 0. This will correspond well with our edge table. This number is the index that is used to retrieve the correct edge intersections. It then calculates the intersections for the vertices.

```
[MaxVertexCount(4)]
void TessellateTetrahedraGS(lineadj SampleData In[4], inout TriangleStream<SurfaceVertex>
Stream)
{
	// First check which vertices are inside.
	// We store this as a bit index
	// Each bit represents yes or no
	uint index =	(uint(In[0].Field.w > 1) << 3) |
			(uint(In[1].Field.w > 1) << 2) |
			(uint(In[2].Field.w > 1) << 1) |
			 uint(In[3].Field.w > 1);
	// Only calculate when we are inside
	if(index > 0 && index < 15)
	{
		uint4 e0 = EdgeTableGS[index].e0;
		uint4 e1 = EdgeTableGS[index].e1;

		Stream.Append(CalculateIntersection(In[e0.x],In[e0.y]));

		Stream.Append(CalculateIntersection(In[e0.z],In[e0.w]));
		Stream.Append(CalculateIntersection(In[e1.x],In[e1.y]));

		// Sometimes we need 4 vertices
		if(e1.z != 0)
		{
			Stream.Append(CalculateIntersection(In[e1.z],In[e1.w]));
		}
	}
}
```

Conclusion

Marching cubes is a fun way of visualizing metaballs although it has its drawbacks. The biggest problem is the need for a grid with a good enough resolution to have nice results. In the engine making the grid too big will result in a bad allocation exception. Maybe instead of creating the entire grid, it's an interesting idea to only create a subsection around our metaballs and update the grid every time a ball moves. In the implementation you can sometimes also see the metaballs trying to go outside the grid. This will cap them, again creating grids locally could solve this. Considering these problems the algorithm is still useful and fast for 3D visualization.

Final Result



Bibliography

Bourke, P. (1994, May). *Polygonising a scalar field*. Retrieved from paulbourke.net: http://paulbourke.net/geometry/polygonise/

Geiss, R. (2000, 10 3). *Metaballs*. Retrieved from geisswerks: http://www.geisswerks.com/ryan/BLOBS/blobs.html

Lingrand, D. (n.d.). *Marching Cubes*. Retrieved from http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html

nvidia. (n.d.). *nvidia metaballs example*. Retrieved from http://developer.download.nvidia.com/SDK/10/direct3d/screenshots/samples/MetaBalls.html